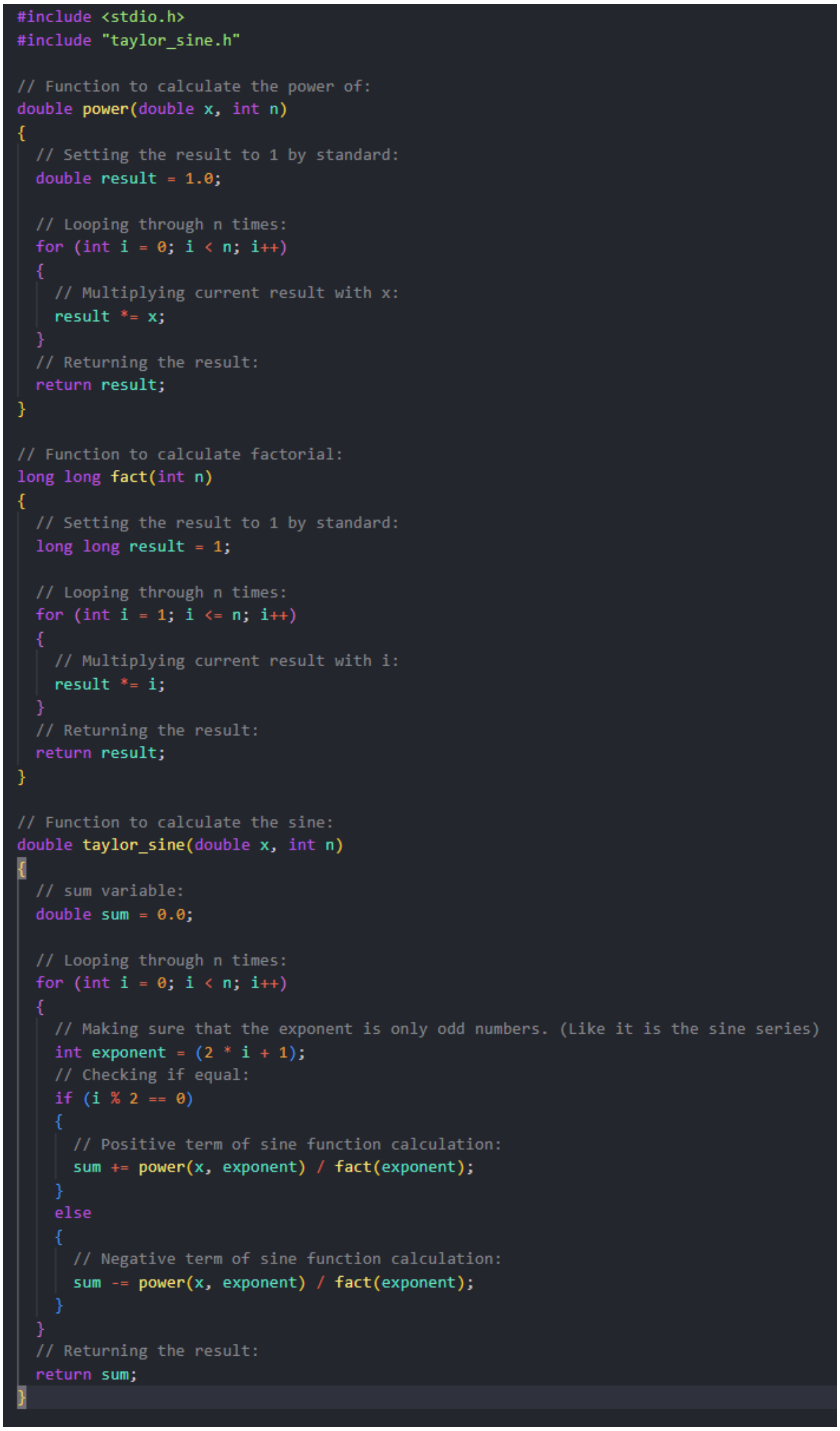
https://github.com/IPFCE-2024/assignment-7-nicolas-anton

Exercise 1:

a) Sine function:

(taylor_sine.c)

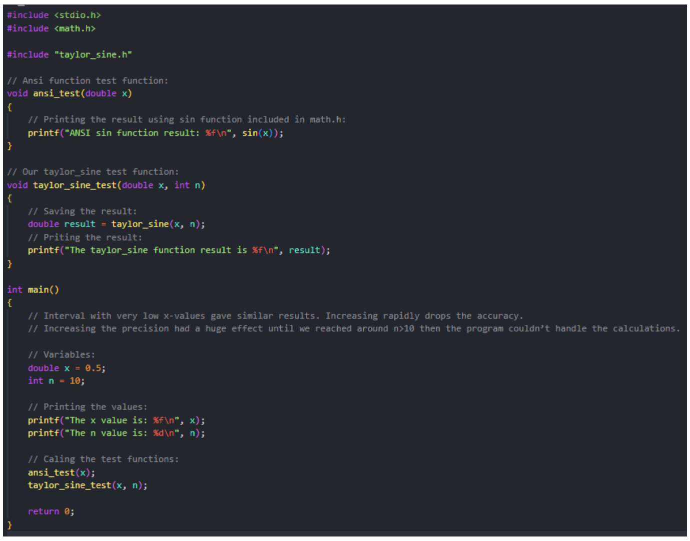


```c
#include <stdio.h>
#include "taylor_sine.h"

// Function to calculate the power of:
double power(double x, int n)
{
  // Setting the result to 1 by standard:
  double result = 1.0;

  // Looping through n times:
  for (int i = 0; i < n; i++)
  {
    // Multiplying current result with x:
    result *= x;
  }
  // Returning the result:
  return result;
}

// Function to calculate factorial:
long long fact(int n)
{
  // Setting the result to 1 by standard:
  long long result = 1;

  // Looping through n times:
  for (int i = 1; i <= n; i++)
  {
    // Multiplying current result with i:
    result *= i;
  }
  // Returning the result:
  return result;
}

// Function to calculate the sine:
double taylor_sine(double x, int n)
{
  // sum variable:
  double sum = 0.0;

  // Looping through n times:
  for (int i = 0; i < n; i++)
  {
    // Making sure that the exponent is only odd numbers. (Like it is the sine series)
    int exponent = (2 * i + 1);
    // Checking if equal:
    if (i % 2 == 0)
    {
      // Positive term of sine function calculation:
      sum += power(x, exponent) / fact(exponent);
    }
    else
    {
      // Negative term of sine function calculation:
      sum -= power(x, exponent) / fact(exponent);
    }
  }
  // Returning the result:
  return sum;
}
```

b) Write some tests for different values of x (try both small and large input values), and compare your function output with the ANSI C sin function.

Test program:

(test_file.c)



```c
#include <stdio.h>
#include <math.h>

#include "taylor_sine.h"

// Ansi function test function:
void ansi_test(double x)
{
    // Printing the result using sin function included in math.h:
    printf("ANSI sin function result: %f\n", sin(x));
}

// Our taylor_sine test function:
void taylor_sine_test(double x, int n)
{
    // Saving the result:
    double result = taylor_sine(x, n);
    // Priting the result:
    printf("The taylor_sine function result is %f\n", result);
}

int main()
{
    // Interval with very low x-values gave similar results. Increasing rapidly drops the accuracy.
    // Increasing the precision had a huge effect until we reached around n>10 then the program couldn’t handle the calculations.

    // Variables:
    double x = 0.5;
    int n = 10;

    // Printing the values:
    printf("The x value is: %f\n", x);
    printf("The n value is: %d\n", n);

    // Caling the test functions:
    ansi_test(x);
    taylor_sine_test(x, n);

    return 0;
}
```

Tests:

taylor_sine.c (n = 8):

x	Output
0	0.000000
0.5	0.479426
5	-0.960921
-5	0.960921
100	-7489051144455195136.000000

test_file.c:

x	Output
0	0.000000
0.5	0.479426
5	-0.958924
-5	0.958924
100	-0.506366

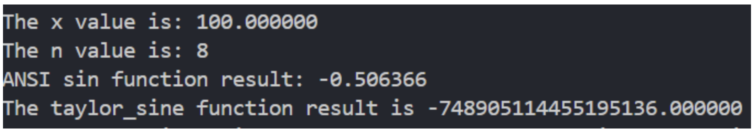
We can see that the accuracy of the taylor_sine.c function is perfect at very small x-values and actually at small negative values as well. Immediately when we raise the value the accuracy is completely off. The test file makes perfect calculations.

c) Which intervals of input x did your function give a similar result to the ANSI C sin function? What impact did increasing the precision have (i.e. increasing the number of Taylor series terms)?

Interval with very low x-values gave similar results. Increasing rapidly drops the accuracy.

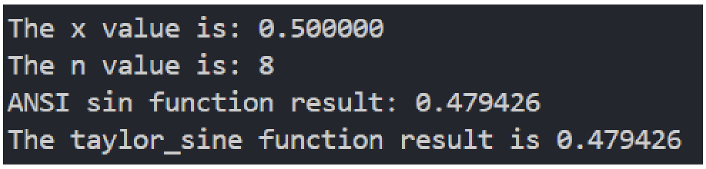
Increasing the precision had a huge positive effect until we reach very high n-values then the program couldn't handle the calculations possibly due to overflow.

At $x = 100$ and $n = 8$:



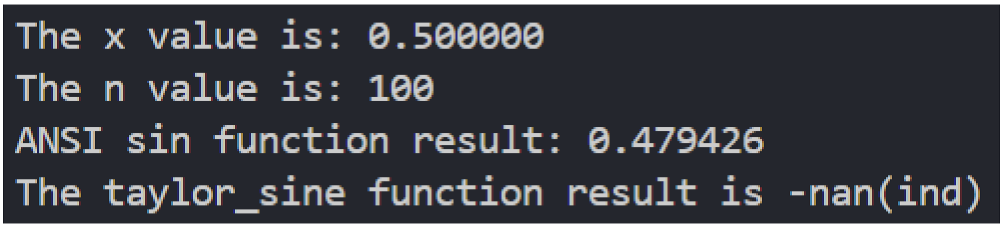
```
The x value is: 100.000000
The n value is: 8
ANSI sin function result: -0.506366
The taylor_sine function result is -748905114455195136.000000
```

At $x = 0.5$ and $n = 8$:



```
The x value is: 0.500000
The n value is: 8
ANSI sin function result: 0.479426
The taylor_sine function result is 0.479426
```

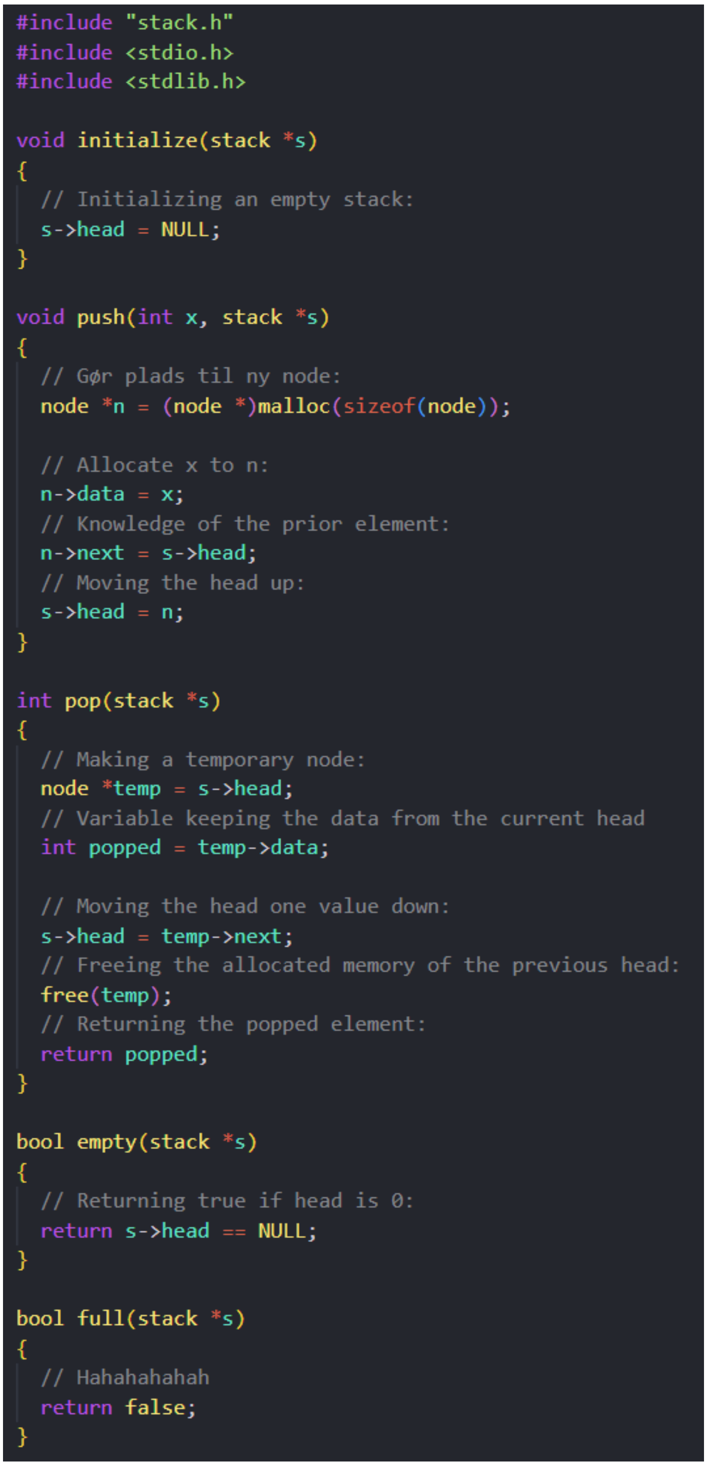
Showing the overflow with high precision:



```
The x value is: 0.500000
The n value is: 100
ANSI sin function result: 0.479426
The taylor_sine function result is -nan(ind)
```

Exercise 2:

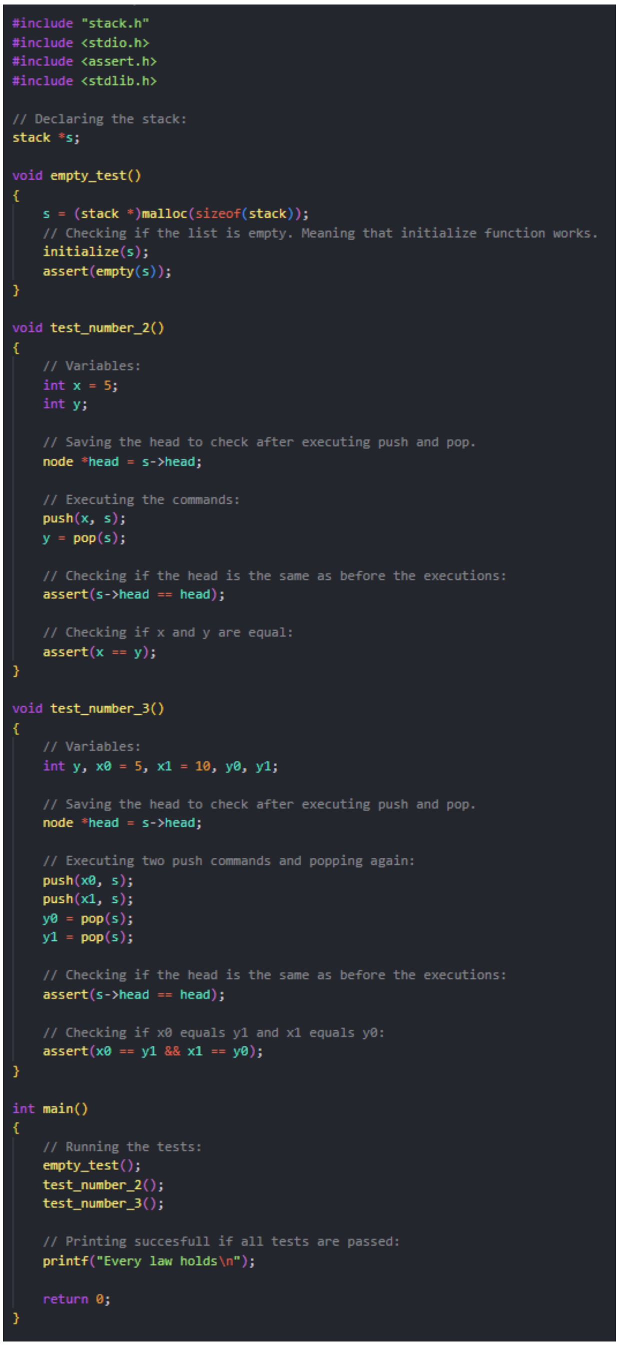
a) Implement a stack based on singly-linked lists as discussed in the lecture.



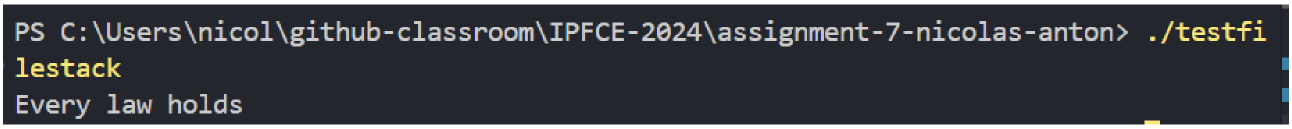
```c
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>

void initialize(stack *s)
{
  // Initializing an empty stack:
  s->head = NULL;
}

void push(int x, stack *s)
{
  // Gør plads til ny node:
  node *n = (node *)malloc(sizeof(node));

  // Allocate x to n:
  n->data = x;
  // Knowledge of the prior element:
  n->next = s->head;
  // Moving the head up:
  s->head = n;
}

int pop(stack *s)
{
  // Making a temporary node:
  node *temp = s->head;
  // Variable keeping the data from the current head
  int popped = temp->data;

  // Moving the head one value down:
  s->head = temp->next;
  // Freeing the allocated memory of the previous head:
  free(temp);
  // Returning the popped element:
  return popped;
}

bool empty(stack *s)
{
  // Returning true if head is 0:
  return s->head == NULL;
}

bool full(stack *s)
{
  // Hahahahahah
  return false;
}
```

b) Testing:

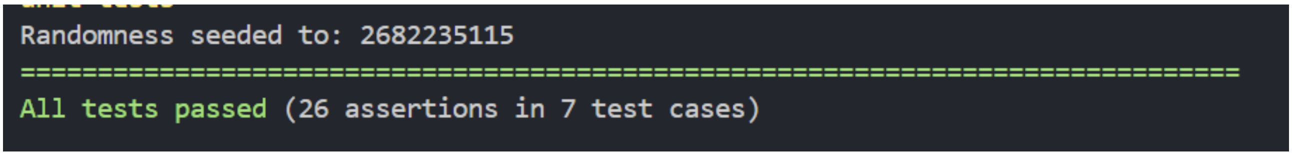
“testfilestack.c”:



```c
#include "stack.h"
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

// Declaring the stack:
stack *s;

void empty_test()
{
    s = (stack *)malloc(sizeof(stack));
    // Checking if the list is empty. Meaning that initialize function works.
    initialize(s);
    assert(empty(s));
}

void test_number_2()
{
    // Variables:
    int x = 5;
    int y;

    // Saving the head to check after executing push and pop.
    node *head = s->head;

    // Executing the commands:
    push(x, s);
    y = pop(s);

    // Checking if the head is the same as before the executions:
    assert(s->head == head);

    // Checking if x and y are equal:
    assert(x == y);
}

void test_number_3()
{
    // Variables:
    int y, x0 = 5, x1 = 10, y0, y1;

    // Saving the head to check after executing push and pop.
    node *head = s->head;

    // Executing two push commands and popping again:
    push(x0, s);
    push(x1, s);
    y0 = pop(s);
    y1 = pop(s);

    // Checking if the head is the same as before the executions:
    assert(s->head == head);

    // Checking if x0 equals y1 and x1 equals y0:
    assert(x0 == y1 && x1 == y0);
}

int main()
{
    // Running the tests:
    empty_test();
    test_number_2();
    test_number_3();

    // Printing succesfull if all tests are passed:
    printf("Every law holds\n");

    return 0;
}
```



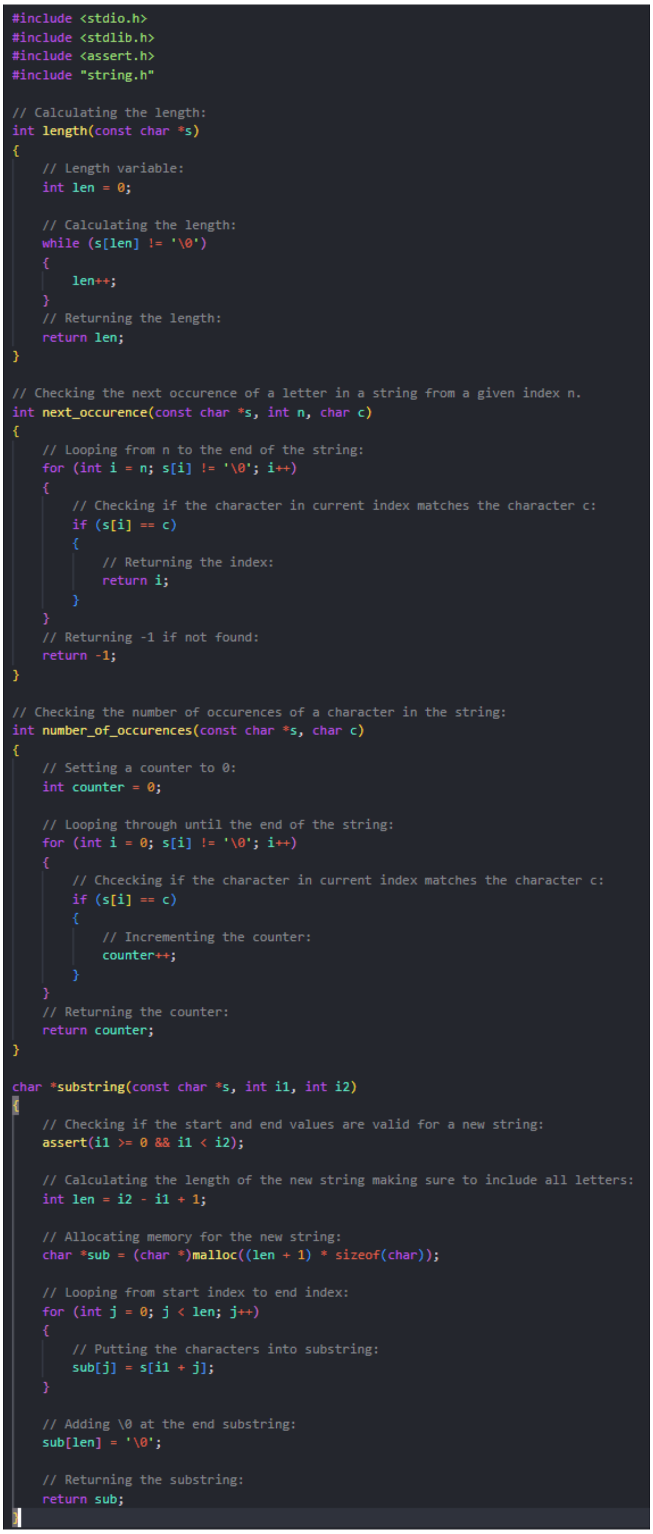
```
PS C:\Users\nicol\github-classroom\IPFCE-2024\assignment-7-nicolas-anton> ./testfi
lestack
Every law holds
```



```
Randomness seeded to: 2682235115
===============================================================================
All tests passed (26 assertions in 7 test cases)
```

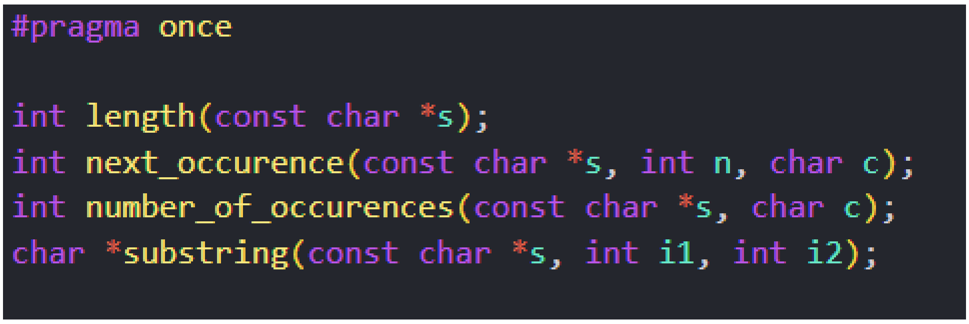
Optional (alle pånær sidste opgave):

string.c:



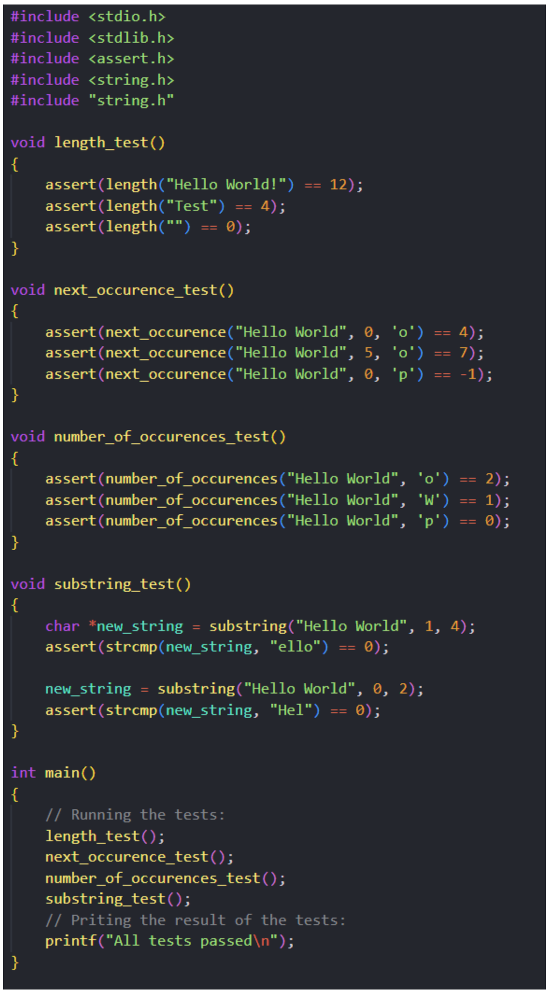
```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "string.h"

// Calculating the length:
int length(const char *s)
{
    // Length variable:
    int len = 0;

    // Calculating the length:
    while (s[len] != '\0')
    {
        len++;
    }
    // Returning the length:
    return len;
}

// Checking the next occurence of a letter in a string from a given index n.
int next_occurence(const char *s, int n, char c)
{
    // Looping from n to the end of the string:
    for (int i = n; s[i] != '\0'; i++)
    {
        // Checking if the character in current index matches the character c:
        if (s[i] == c)
        {
            // Returning the index:
            return i;
        }
    }
    // Returning -1 if not found:
    return -1;
}

// Checking the number of occurences of a character in the string:
int number_of_occurences(const char *s, char c)
{
    // Setting a counter to 0:
    int counter = 0;

    // Looping through until the end of the string:
    for (int i = 0; s[i] != '\0'; i++)
    {
        // Chcecking if the character in current index matches the character c:
        if (s[i] == c)
        {
            // Incrementing the counter:
            counter++;
        }
    }
    // Returning the counter:
    return counter;
}

char *substring(const char *s, int i1, int i2)
{
    // Checking if the start and end values are valid for a new string:
    assert(i1 >= 0 && i1 < i2);

    // Calculating the length of the new string making sure to include all letters:
    int len = i2 - i1 + 1;

    // Allocating memory for the new string:
    char *sub = (char *)malloc((len + 1) * sizeof(char));

    // Looping from start index to end index:
    for (int j = 0; j < len; j++)
    {
        // Putting the characters into substring:
        sub[j] = s[i1 + j];
    }

    // Adding \0 at the end substring:
    sub[len] = '\0';

    // Returning the substring:
    return sub;
}
```

string.h:

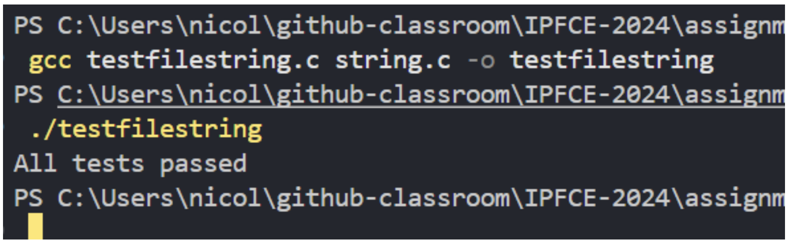


```c
#pragma once

int length(const char *s);
int next_occurence(const char *s, int n, char c);
int number_of_occurences(const char *s, char c);
char *substring(const char *s, int i1, int i2);
```

testfilestring.c:



```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "string.h"

void length_test()
{
    assert(length("Hello World!") == 12);
    assert(length("Test") == 4);
    assert(length("") == 0);
}

void next_occurence_test()
{
    assert(next_occurence("Hello World", 0, 'o') == 4);
    assert(next_occurence("Hello World", 5, 'o') == 7);
    assert(next_occurence("Hello World", 0, 'p') == -1);
}

void number_of_occurences_test()
{
    assert(number_of_occurences("Hello World", 'o') == 2);
    assert(number_of_occurences("Hello World", 'W') == 1);
    assert(number_of_occurences("Hello World", 'p') == 0);
}

void substring_test()
{
    char *new_string = substring("Hello World", 1, 4);
    assert(strcmp(new_string, "ello") == 0);

    new_string = substring("Hello World", 0, 2);
    assert(strcmp(new_string, "Hel") == 0);
}

int main()
{
    // Running the tests:
    length_test();
    next_occurence_test();
    number_of_occurences_test();
    substring_test();
    // Priting the result of the tests:
    printf("All tests passed\n");
}
```



```
PS C:\Users\nicol\github-classroom\IPFCE-2024\assignm
 gcc testfilestring.c string.c -o testfilestring
PS C:\Users\nicol\github-classroom\IPFCE-2024\assignm
 ./testfilestring
All tests passed
PS C:\Users\nicol\github-classroom\IPFCE-2024\assignm
```